Corbin Mayes - 1/31/19

Constraint Satisfaction

Backtracking Solver

To implement the backtracking search, I followed the pseudo-code in the book (Figure 6.5). Because MRV, LCV, and MAC-3 were not in my code for the backtracker yet, I used select_unassigned_variable function in place of MRV which just picks the next possible unassigned variable. Instead of LCV, I call order_domain_variables which returns the entire domain of the csp for the backtracking search to look through. And for MAC-3, I just implemented a no_inference which just returns true.

CSP

I decided to do an overarching CSP class that all of my other CSP's would call to run all of their generalized functions from. The CSP maintains all the assigning and the unassigning of variables as well as calculating the number of conflicts a certain variable and value pairing may have within the current assignment and the goal test to see if all variables have been assigned. Also within the CSP class is all the functions for pruning, assuming, and getting the inferences from a certain variable and value pairing. Because the constraints for each problem is unique to that problem I decided to leave that out of the over-arching CSP class and just pass the function in as a instance variable.

MapColorCSP

Initially for the MapColorCSP, I was planning on using inheritance and have MapColorCSP be a subclass of CSP. However, after struggling a bit with the initialization of the MapColorCSP with calling the initialization of the CSP class, I realized that the MapColorCSP wasn't adding any new variables and the only new function was constraints but that was already being passed into the CSP as a variable. So I added a global variable called csp that is just the CSP with the unique MapColorCSP variables being passed in. The variables and domain both have dictionaries that translate the numerical value to the original strings. The constraints function is just to test whether two variables have the same values because only neighbors of the variable being tested are passed into the second variable spot the constraint doesn't need to test for this constraint. I added the solve function to run the backtracking and get the solution because I wanted to translate the solution back into the original using the dictionaries in the MapColorCSP. This just kept the test codes cleaner and less cluttered by just calling the solve function after defining the class.

CircuitBoardCSP

For the CircuitBoardCSP, I followed a very similar approach to what I did with MapColorCSP. I began by creating a Circuit class because each circuit has multiple variables for defining it so it made sense to store that in one place that was easy to access and call the variables from. From there I initialized the variables and domains' lists and dictionaries. For the domain, I had to iterate over all the x and y combos for possible locations on the board. Once all the variables were created, I created the CSP and built the constraints function. The constraints function iterates through two circuits to test whether or not they intersect as well as are completely on the circuit board. For the solve function, I followed the same approach as with MapColorCSP except I wanted to print out the actual solution as a board. So I created a list of lists to represent the rows and columns and once that was properly filled, I translated it into a string and printed the string.

MRV Heuristic

I initially had problems with importing argmin_random_tie from utils to use in the MRV so I found pseudo-code for it and created my own to use in this assignment. It returns the element with the lost score or breaks the tie if there is one. I also needed to calculate the number of legal moves from the current variable and assignment so I built the function num_legal_moves to do so by either taking from the current domain or to caluculate using the csp's num_conflicts function. The mrv function then creates a list of unassigned variables that it passes onto the argmin_random_tie function and returns the result.

LCV Heuristic

To implement LCV, I just sorted the result from my csp choices function given a certain variable using the num_conflicts function as the deciding factor of how to sort the variables.

MAC-3

Since MAC-3 calls the return of the AC-3 function I began by implementing it to check for arc consistency. I based it off of Figure 6.3 in the book. To run the AC-3, I needed to check if a value was removed and so I added the revise function to do so.

CS1 Section Assignment

For the extra CS1 Section Assignment, I followed my other CSP's outline with the init, constraints, and solve functions. I began by creating a person class that holds their name, availability and whether or not they are section leader for the same reasons as the CircuitBoardCSP. Creating the variables' list and dictionary was trickier because I was actually passing in the list of section leaders with their available times and a separate list of students with their available times as well as a list of all possible sections. I began by iterating and adding the section leaders to the variables and then iterate and add the students afterwards. All the sections would become the domain and everything would be passed into the CSP. To build the constraints function, I followed the constraints outlined in the assignment. First, I made sure that the time being tested is in the person's availability. Next, I made sure that if both the variables are section leaders that their times don't match i.e. there isn't multiple section leaders per section. I had to make a change for this constraints function and also pass assignment into it so I could check to make sure that all the sections had a proper number of students and also that each section of students had a section leader in it. The solve function is almost identical to the MapColorCSP. Because the class actually takes input, and I didn't have enough time to create text files and implement the code to translate those, I just passed some lists that are written in the test function to create a section schedule.